return-Oriented Programming

David Brumley
Carnegie Mellon University

Credit: Some slides from Ed Schwartz
Control Flow Hijack: Always control + computation

shellcode (aka payload)  padding  &buf

computation + control

Return-oriented programming (ROP): shellcode without code injection
Motivation: Return-to-libc Attack

Overwrite return address with address of libc function
• setup fake return address and argument(s)
• ret will “call” libc function

No injected code!
What if we don’t know the absolute address any pointers to “/bin/sh”

(objdump gives addresses, but we don’t know ASLR constants)
Need to find an instruction sequence, aka *gadget*, with esp

ptr to "/bin/sh"
&system
caller’s ebp
buf (64 bytes)
argv[1]
buf
%esp
%ebp
Scorecard for ret2libc

• No injected code ➔ DEP ineffective

• Requires knowing address of system

• ... or does it.
Return Oriented Programming Techniques

1. Geometry of Flesh on the Bone, Shacham et al, CCS 2007
ROP Programming

1. Disassemble code
2. Identify *useful* code sequences as gadgets
3. Assemble gadgets into desired shellcode
There are many *semantically equivalent* ways to achieve the same net shellcode effect.
Equivalence

Mem[v2] = v1

Desired Logic

\[ a_1: \text{mov eax, [esp]} \]
\[ a_2: \text{mov ebx, [esp+8]} \]
\[ a_3: \text{mov [ebx], eax} \]

Implementation 1

Stack

\[
\begin{align*}
\text{...} \\
\text{v}_2 \\
\text{...} \\
\text{v}_1 \\
\text{esp}
\end{align*}
\]
Gadgets

A gadget is any instruction sequence ending with `ret`
Return-Oriented Programming

is a lot like a ransom note, but instead of cutting cut letters from magazines, you are cutting out instructions from text segments.
ROP Overview

• Idea: We forge shell code out of existing application logic gadgets

• Requirements: vulnerability + gadgets + some unrandomized code

• History:
  – No code randomized: Code injection
  – DEP enabled by default: ROP attacks using libc gadgets publicized ~2007
  – Libc randomized
  – ASLR library load points
  – Q builds ROP compiler using .text section
  – Today: Windows 7 compiler randomizes text by default, Randomizing text on Linux not straight-forward.
Gadgets

Desired Logic

Mem[v2] = v1

Stack

Suppose a_2 and a_3 on stack

<table>
<thead>
<tr>
<th>eax</th>
<th>v_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td></td>
</tr>
<tr>
<td>eip</td>
<td>a_1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_2</td>
</tr>
<tr>
<td>a_3</td>
</tr>
<tr>
<td>v_1</td>
</tr>
</tbody>
</table>

Implementation 2

a_1: pop eax;
a_2: ret
a_3: pop ebx;
a_4: ret
a_5: mov [ebx], eax
Gadgets

Desired Logic

\[
\text{Mem}[v2] = v1
\]

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_5)</td>
</tr>
<tr>
<td>(v_2)</td>
</tr>
<tr>
<td>(a_3)</td>
</tr>
<tr>
<td>(v_1)</td>
</tr>
</tbody>
</table>

Implementation 2

- \(a_1\): pop eax;
- \(a_2\): ret
- \(a_3\): pop ebx;
- \(a_4\): ret
- \(a_5\): mov [ebx], eax
Gadgets

Desired Logic

Mem[v2] = v1

<table>
<thead>
<tr>
<th>eax</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>v2</td>
</tr>
<tr>
<td>eip</td>
<td>a3</td>
</tr>
</tbody>
</table>

Stack

<table>
<thead>
<tr>
<th>a5</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2</td>
</tr>
<tr>
<td>a3</td>
</tr>
<tr>
<td>v1</td>
</tr>
</tbody>
</table>

Implementation 2

a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
Gadgets

Desired Logic

\[ \text{Mem}[v2] = v1 \]

<table>
<thead>
<tr>
<th>eax</th>
<th>( v_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>( v_2 )</td>
</tr>
<tr>
<td>eip</td>
<td>( a_g )</td>
</tr>
</tbody>
</table>

Implementation 2

\[ a_1: \text{pop eax}; \]
\[ a_2: \text{ret} \]
\[ a_3: \text{pop ebx}; \]
\[ a_4: \text{ret} \]
\[ a_5: \text{mov [ebx], eax} \]
**Gadgets**

**Desired Logic**

\[
\text{Mem}[v2] = v1
\]

<table>
<thead>
<tr>
<th>eax</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>v2</td>
</tr>
<tr>
<td>eip</td>
<td>a5</td>
</tr>
</tbody>
</table>

**Stack**

\[
\begin{align*}
\text{esp} & \\
a_5 & \\
v_2 & \\
a_3 & \\
v_1 &
\end{align*}
\]

**Implementation 2**

\[
\begin{align*}
a_1 & : \text{pop eax;} \\
a_2 & : \text{ret} \\
a_3 & : \text{pop ebx;} \\
a_4 & : \text{ret} \\
a_5 & : \text{mov [ebx], eax}
\end{align*}
\]
Equivalence

Mem[\(v_2\)] = \(v_1\)

Desired Logic

\[\begin{array}{c}
a_3 \\
v_2 \\
a_2 \\
v_1 \\
\end{array}\]

Stack

Implementation 1

\[\begin{array}{c}
a_1: pop eax; ret \\
a_2: pop ebx; ret \\
a_3: mov [ebx], eax \\
\end{array}\]

Implementation 2

"Gadgets"
Return-Oriented Programming (ROP)

Desired Shellcode

- Find needed instruction gadgets at addresses a₁, a₂, and a₃ in existing code
- Overwrite stack to execute a₁, a₂, and then a₃

Mem[v2] = v1
Return-Oriented Programming (ROP)

Mem[v2] = v1

Desired Shellcode

\[\text{argv} \[0\] \text{ Mem[argv]} \]

\[\text{argv} \[1\] \text{ buf} \]

\[\text{caller's ebp} \]

\[\text{return} \]

\[\text{24 bytes} \]

Desired store executed!
void foo(char *input) {
    char buf[512];
    ...
    strcpy (buf, input);
    return;
}

a₁: add eax, 0x80; pop %ebp; ret
a₂: pop %eax; ret

ret at a₃

Draw a stack diagram and ROP exploit to pop a value 0xBBBBBBBBBB into eax and add 80.
void foo(char *input) {
    char buf[512];
    ...
    strcpy(buf, input);
    return;
}

a₁: add eax, 0x80; pop %ebp; ret
a₂: pop %eax; ret

ret at a₃

<data for pop ebp>
a₃
0xBBB BBB BBB
a₂
a₃
saved ebp
buf

Overwrite buf
Start rop chain
gadget 1 + data
gadget 2

AAAAA ... a₃ a₂ 0xBBB BBB BBB a₁
Example in 04-exercises
Attack Surface: Linux

Unrandomized

Program Image

Randomized

Libc

Stack

Heap
Attack Surface: Windows

Unrandomized

Program Image

Libc

Randomized

Stack

Heap
Gadget 1
push %esp
mov %eax, %edx
pop %edi
ret

Gadget 2
push %edi
pop %eax
pop %ebp
ret

Useful, for example, to get a copy of ESP. If we know relative offset of ptr to esp, we can know use that relative offset knowledge to locate a pointer. (e.g., find more gadgets that add offset and store result to stack.)

This overcomes ASLR because ASLR only protects against knowing absolute addresses.
LPVOID WINAPI VirtualProtect(
    LPVOID lpAddress, // base addr to pages to change
    SIZE_T dwSize, // size of the region in bytes
    DWORD flNewProtect, // 0x40 = EXECUTE_READWRITE
    DWORD flProtect // A ptr to a variable for prev. arg
);

VirtualProtect() to un-DEP memory region
Practical Matters

• Stack pivots: point esp at heap, e.g., because we control heap data.

• See
Disassembling Code
Recall: Execution Model

- **Process Memory**
  - **Code**
  - **Stack**
  - **Heap**

**Processor**
- **EIP**

Flows:
- **Fetch, decode, execute**
- **read and write**
## Disassembly

```
user@box:~/l2$ objdump -d ./file
...
00000000 <even_sum>:
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>1:</td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>3:</td>
<td>83 ec 10</td>
<td>sub $0x10,%esp</td>
</tr>
<tr>
<td>6:</td>
<td>8b 45 0c</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>9:</td>
<td>03 45 08</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>c:</td>
<td>03 45 10</td>
<td>add 0x10(%ebp),%eax</td>
</tr>
<tr>
<td>f:</td>
<td>89 45 fc</td>
<td>mov %eax,0xfffffffffc(%ebp)</td>
</tr>
<tr>
<td>12:</td>
<td>8b 45 fc</td>
<td>mov 0xfffffffffc(%ebp),%eax</td>
</tr>
<tr>
<td>15:</td>
<td>83 e0 01</td>
<td>and $0x1,%eax</td>
</tr>
<tr>
<td>18:</td>
<td>84 c0</td>
<td>test %al,%al</td>
</tr>
<tr>
<td>1a:</td>
<td>74 03</td>
<td>je 1f &lt;even_sum+0x1f&gt;</td>
</tr>
<tr>
<td>1c:</td>
<td>ff 45 fc</td>
<td>incl 0xfffffffffc(%ebp)</td>
</tr>
<tr>
<td>1f:</td>
<td>8b 45 fc</td>
<td>mov 0xfffffffffc(%ebp),%eax</td>
</tr>
<tr>
<td>22:</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>23:</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

---

**Disassemble**

**Address**

**Executable instructions**
Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Algorithm:
1. Decode Instruction
2. Advance EIP by len
Linear-Sweep Disassembly

Executable Instructions

0x55  0x89  0xe5  0x83  0xec  0x10  ...  0xc9

Disassembler

**EIP**

Disassembler Instructions

push ebp
mov %esp, %ebp

**Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte**

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>Mod</th>
<th>R/M</th>
<th>Value of ModR/M Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(EAX)</td>
<td>00</td>
<td>008</td>
<td>10, 18, 20, 28, 30, 38</td>
</tr>
<tr>
<td>(ECX)</td>
<td>001</td>
<td>019</td>
<td>11, 19, 21, 29, 31, 39</td>
</tr>
</tbody>
</table>

...
Linear-Sweep Disassembly

Executable Instructions
0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassembler
EIP

Algorithm:
1. Decode Instruction
2. Advance EIP by len

Note we don't follow jumps: we just increment by instruction length

push ebp
mov %esp, %ebp
Disassemble from any address

It’s perfectly valid to start disassembling from *any* address.
All byte sequences will have a unique disassembly
Recursive Descent

• Follow jumps and returns instead of linear sweep

• Undecidable: indirect jumps
  – Where does jmp *eax go?
ROP Programming

1. Disassemble code
2. Identify *useful* code sequences ending in ret as gadgets
3. Assemble gadgets into desired shellcode
Gadgets, Historically

• Shacham et al. manually identified which sequences ending in ret in libc were useful gadgets
• Common shellcode was created with these gadgets.
• Everyone used libc, so gadgets and shellcode universal

Semantics

\[
\begin{align*}
\text{Mem}[v2] &= v1 \\
a_1: & \text{ pop eax; ret} \\
\ldots & \\
a_3: & \text{ mov [ebx], eax} \\
\ldots & \\
a_2: & \text{ pop ebx; ret}
\end{align*}
\]
ROP: Shacham et al.

1. Disassemble code
2. Identify *useful* code sequences as gadgets ending in `ret`
3. Assemble gadgets into desired shellcode

Then Q came along and automated
END
Q: Automating ROP

Overview*

* Exploit hardening step not discussed here.
Step 1: Disassemble code

Step 2: Identify *useful* code sequences (not necessarily ending in `ret`)

\[ \text{"useful"} = Q - Op \]
# Q-Ops (aka Q Semantic Types)
(think instruction set architecture)

<table>
<thead>
<tr>
<th>Q-Op</th>
<th>Semantics</th>
<th>Real World Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>MoveRegG($t_1$, $t_2$)</td>
<td>$t_1 := t_2$</td>
<td>xchg %eax, %ebp; ret</td>
</tr>
<tr>
<td>LoadConstG($t_1$, $c$)</td>
<td>$t_1 := c$</td>
<td>pop %ebp; ret</td>
</tr>
<tr>
<td>ArithmeticG($t_1$, $t_2$, $t_3$, $op$)</td>
<td>$t_1 := t_2 \ op t_3$;</td>
<td>add %edx, %eax; ret</td>
</tr>
<tr>
<td>LoadMemG($t_1$, $t_2$, $c$)</td>
<td>$t_1 := [t_2 + c]$</td>
<td>movl 0x60(%eax), %eax; ret</td>
</tr>
<tr>
<td>StoreMemG($t_1$, $c$, $t_2$)</td>
<td>$[t_1+c] := t_2$</td>
<td>mov %dl, 0x13(%eax); ret</td>
</tr>
<tr>
<td>ArithmeticLoadG($t_1$, $t_2$, $c$, $op$)</td>
<td>$t_1 := t_1 \ op [t_2 + c]$</td>
<td>add0x1376db4(%ebx), %ecx; (...); ret</td>
</tr>
<tr>
<td>ArithmeticStoreG($t_1$, $t_2$, $c$, $op$)</td>
<td>$[t_1+c] := [t_1+c] \ op t_2$</td>
<td>add %al, 0x5de474c0(%ebp); ret</td>
</tr>
</tbody>
</table>
Q-Ops (aka Q Semantic Types) (think instruction set architecture)

<table>
<thead>
<tr>
<th>Q-Op</th>
<th>Semantics</th>
<th>Real World Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>MoveRegG(t₁, t₂)</td>
<td>( t₁ := t₂ )</td>
<td>xchg %eax, %ebp; ret</td>
</tr>
<tr>
<td>LoadConstG(t₁, c)</td>
<td>( t₁ := c )</td>
<td>pop %ebp; ret</td>
</tr>
<tr>
<td>ArithmeticG(t₁, t₂, t₃, op)</td>
<td>( t₁ := t₂ )( op ) ( t₃ )</td>
<td>add %edx, %eax; ret</td>
</tr>
<tr>
<td>LoadMemG(t₁, t₂, c)</td>
<td>( t₁ := [t₂ + c] )</td>
<td>movl 0x60(,%eax), %eax; mov %eax, 0x1376dabe4(%ebx), %ecx; (...) ; ret</td>
</tr>
<tr>
<td>StoreMemG(t₁, c, t₂)</td>
<td>( [t₁ + c] := t₂ )</td>
<td>mov %dl, 0x1376dabe4(%ebp); ret</td>
</tr>
<tr>
<td>ArithmeticLoadG(t₁, t₂, c, op)</td>
<td>( t₁ := t₁ )( op ) ( [t₂ + c] )</td>
<td>add %al, 0x1376dabe4(%ebx), %ecx; (...) ; ret</td>
</tr>
<tr>
<td>ArithmeticStoreG(t₁, t₂, c, op)</td>
<td>( [t₁ + c] := [t₁ + c] )( op ) ( t₂ )</td>
<td>add %al, 0x1376dabe4(%ebx), %ecx; (...) ; ret</td>
</tr>
</tbody>
</table>

This is not RISC: more Q-Ops gives more opportunities later.

Must be careful attackers, e.g., give c-60 to get c
• Randomized testing tells us we *likely* found a gadget that implements a Q-Op
  – Fast: filters out many candidates
  – Enables more expensive second stage

Executable Code

Linear sweep @ all offsets

Randomized testing of semantics

Prove semantics

Gadget Database
Randomized Testing Example

Before Simulation

What does this do?

EAX 0x0298a7bc
CF 0x1
ESP 0x81e4f104

After Simulation

EAX 0x1
ESP 0x81e4f108
CF 0x1

Semantically
EAX := CF
(MoveRegG)

Probably

sbb %eax, %eax;
neg %eax; ret
Executable Code

Linear sweep @ all offsets

Randomized testing of semantics

Prove semantics

Gadget Database

Turn **probably** into a **proof** that a gadget implements a Q-Op
Proving equivalence

sbb %eax, %eax;  
neg %eax;  ret

Assembly

eax := eax-(eax+CF)  
eax := -eax  
esp := esp+4

BAP

Weakest Precondition

Semantics Gadget

EAX := CF

Yes/No

Prover

Weakest Precondition:

\((eax = eax-(eax+CF)  
eax = -eax  
esp = esp+4)  
\Rightarrow  
(EAX = CF)\)
Proving Equivalence

• Weakest precondition [Dijkstra76] is an algorithm for reducing a program to a statement in logic
  – Q uses predicate logic

• Satisfiability Modulo Theories (SMT) solver, a “Decision Procedure”, determine if statement is true
  – true $\Rightarrow$ semantic gadget
  – Note: “Theorem prover” = undecidable, “SAT solver” = propositional logic

• WP details not discussed here. (It’s a textbook verification technique)
Executable Code

Linear sweep @ all offsets

Randomized testing of semantics

Prove semantics

Gadget Database

✓ Disassemble code
✓ Identify *useful* code sequences as gadgets
• Assemble gadgets into desired shellcode
# Q-Op Gadget Examples

<table>
<thead>
<tr>
<th>Q-Op</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eax := value</code></td>
<td><code>pop %ebp; ret; xchg %eax, %ebp; ret</code></td>
</tr>
<tr>
<td><code>ebx := value</code></td>
<td><code>pop %ebx; pop %ebp; ret</code></td>
</tr>
<tr>
<td>`[ebx +0x5e5b3cc4] := [ebx + offset]</td>
<td>al`</td>
</tr>
<tr>
<td><code>eax := value</code></td>
<td><code>pop %ebp; ret; xchg %eax, %ebp; ret</code></td>
</tr>
<tr>
<td><code>ebp := value</code></td>
<td><code>pop %ebp; ret</code></td>
</tr>
<tr>
<td><code>[ebp + 0xf3774ff] := [ebp + offset] + al</code></td>
<td><code>add %al,0xf3774ff(%ebp); movl $0x85, %dh; ret</code></td>
</tr>
</tbody>
</table>

**apt-get Gadgets**

**Note: extra side-effects handled by Q**
Executable Code

Linear sweep @ all offsets

Randomized testing of semantics

Prove semantics

Gadget Database

QooL Program

Q-Op Arrangement

Gadget Assignment

ROP Shellcode
QooL Language

Motivation:
Write shellcode in high-level language, not assembly

\[
\begin{align*}
\langle \text{exp} \rangle & ::= \text{LOADMEM} \langle \text{exp} \rangle \langle \text{type} \rangle \\
& \quad | \text{BINOP} \langle \text{optype} \rangle \langle \text{exp} \rangle \langle \text{exp} \rangle \\
& \quad | \text{CONST} \langle \text{exp} \rangle \langle \text{type} \rangle \\
\langle \text{stmt} \rangle & ::= \text{STOREMEM} \langle \text{exp} \rangle \langle \text{exp} \rangle \langle \text{type} \rangle \\
& \quad | \text{ASSIGN} \langle \text{var} \rangle \langle \text{exp} \rangle \\
& \quad | \text{CALLEXTERNAL} \langle \text{func} \rangle \langle \text{exp list} \rangle \\
& \quad | \text{SYSCALL}
\end{align*}
\]

QooL Syntax
Example

\[\text{\langle exp \rangle ::= LOADMEM \langle exp \rangle \langle type \rangle} \]
\[| \text{BINOP \langle optype \rangle \langle exp \rangle \langle exp \rangle} \]
\[| \text{CONST \langle exp \rangle \langle type \rangle} \]

\[\text{\langle stmt \rangle ::= STOREMEM \langle exp \rangle \langle exp \rangle \langle type \rangle} \]
\[| \text{ASSIGN \langle var \rangle \langle exp \rangle} \]
\[| \text{CALLEXTERNAL \langle func \rangle \langle exp list \rangle} \]
\[| \text{SYSCALL} \]

Semantics

\[f = \text{LoadMem}[\text{got execve offset}]\]
\[f("/bin/sh") \]

QooL Program

\[f = \text{LoadMem}[\text{got execve offset}]\]
\[\text{arg} = "/bin/sh" \text{ (in hex)}\]
\[\text{StoreMem}(t, \text{adrr})\]
\[f(\text{addr})\]
Q-Op Arrangement

QooL Program

Q-Op Arrangement

Gadget Assignment

\[
\langle \text{exp} \rangle \ ::= \ \text{LOADMEM} \langle \text{exp} \rangle \langle \text{type} \rangle \\
| \ \text{BINOP} \langle \text{optype} \rangle \langle \text{exp} \rangle \langle \text{exp} \rangle \\
| \ \text{CONST} \langle \text{exp} \rangle \langle \text{type} \rangle \\
\langle \text{stmt} \rangle \ ::= \ \text{STOREMEM} \langle \text{exp} \rangle \langle \text{exp} \rangle \langle \text{type} \rangle \\
| \ \text{ASSIGN} \langle \text{var} \rangle \langle \text{exp} \rangle \\
| \ \text{CALLEXTERNAL} \langle \text{func} \rangle \langle \text{exp \ list} \rangle \\
| \ \text{syscall} \\
\]

MoveRegG | LoadConstG \\
| ArithmeticG | LoadMemG \\
| StoreMemG \\
| ArithmeticLoadG \\
| ArithmeticStoreG

Analogy:
Compiling C down to assembly
Every-Munch Algorithm

- **Every Munch**: Conceptually compile QooL program into a set of Q-Op programs
  - Each member tries to use different Q-Ops for the same high-level instructions

- Analogy: Compile C statement to a set of assembly instructions
  - C: \( a = a \times 2; \)
  - Assembly: \( a = a \times 2; \)
    \[ a = a \ll 1; \]
    \[ a = a + a; \]
QooL

- Ultimately pick the smallest Q-Op program that has corresponding gadgets in the target program

- Optimization: Q uses lazy evaluation so programs generated on demand
Assignment chooses a single Q-Op program using real gadgets and register names

*Analogy: Register assignment in a compiler*
Example

Legend

Q-Op
Assembly Gadget

Conflic: %ebx and %ecx mismatch
Example

\[
t_1 := v_1
\]

\[
t_2 := v_2
\]

\[
[t_1] := t_2
\]

\[
\text{mov} \ [eax], \ ebx
\]

\[
\text{ret}
\]

Legend

Q-Op
Assembly Gadget
Recap

Executable Code

Linear sweep @ all offsets

Randomized testing of semantics

Prove semantics

QooL Program

Q-Op Arrangement

Gadget Assignment

Gadget Database

ROP Shellcode
## Real Exploits

Q ROP’ed (and hardened) 9 exploits

<table>
<thead>
<tr>
<th>Name</th>
<th>Total Time</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free CD to MP3 Converter</td>
<td>130s</td>
<td>Windows 7</td>
</tr>
<tr>
<td>Fatplayer</td>
<td>133s</td>
<td>Windows 7</td>
</tr>
<tr>
<td>A-PDF Converter</td>
<td>378s</td>
<td>Windows 7</td>
</tr>
<tr>
<td>A-PDF Converter (SEH exploit)</td>
<td>357s</td>
<td>Windows 7</td>
</tr>
<tr>
<td>MP3 CD Converter Pro</td>
<td>158s</td>
<td>Windows 7</td>
</tr>
<tr>
<td>rsync</td>
<td>65s</td>
<td>Linux</td>
</tr>
<tr>
<td>opendchub</td>
<td>225s</td>
<td>Linux</td>
</tr>
<tr>
<td>gv</td>
<td>237s</td>
<td>Linux</td>
</tr>
<tr>
<td>Proftpd</td>
<td>44s</td>
<td>Linux</td>
</tr>
</tbody>
</table>
ROP Probability

• Given program size, what is the probability Q can create a payload?
  – Measure over all programs in /usr/bin

• Depends on target computation
  – Call libc function in GOT
  – Call libc function not in GOT
ROP Probability

Program Size (bytes)

Probability that attack works

Call libc functions in 80% of programs >= true (20KB)
Q ROP Limitations

• Q’s gadgets types are not Turing-complete
  – Calling system("/bin/sh") or mprotect() usually enough
  – Shacham showed libc has a Turing-complete set of gadgets.

• Q does not find conditional gadgets
  – Potential automation of interesting work on ROP without Returns [CDSSW10]

• Q does not minimize ROP payload size
Research Summary

1. Disassemble code
2. Identify *useful* code sequences as gadgets
3. Assemble gadgets into desired shellcode

Shachem: Automatic

Shacham: Manual, Turing-complete

Q: Automatic, not Turing complete
Backup slides here.

• Titled cherries because they are for the pickin. (credit due to maverick for wit)
Stencils

ABC

ABC

ABC

ABC

ABC

ABC

ABC

ABC

ABC

ABC
Other Colors from Adobe Kuler

Don’t use these unless absolutely necessary. We are not making skittles, so there is no rainbow of colors necessary.

Mac application for Adobe Kuler:
http://www.lithoglyph.com/mondrianum/
http://kuler.adobe.com/